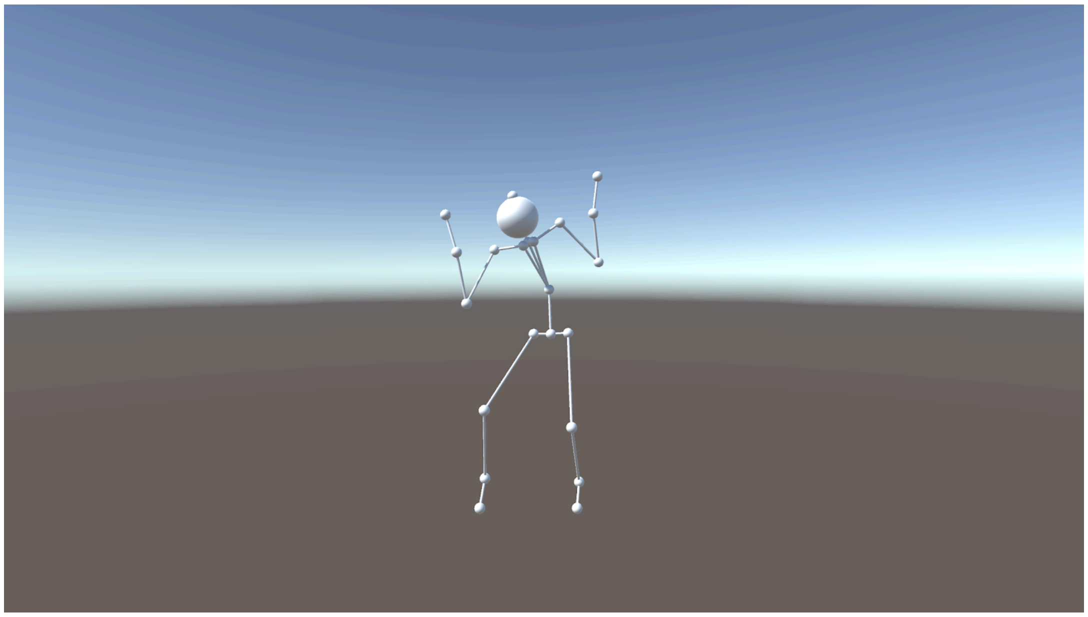
Exercise 1 - Animation & Transformations

In this exercise you will use BVH files to draw and animate a 3D character on screen.

The goal of this exercise is to learn about animation, homogeneous coordinates and 3D transformation matrices, as well as quaternions.

You **must** submit this exercise in pairs.



EX1 Guidelines

- In this exercise you may only edit the files CharacterAnimator.cs and QuaternionUtils.cs, according to the instructions.
- You are given 2 helper files - BVHParser.cs and MatrixUtils.cs. Their documentation is provided at the end of this file. You may not edit or change these files in any way.
- You are also given a folder of different BVH files which you can use to check your code.

3. **Important!** In this exercise you may not use any function of `GameObject.transform` or directly set any of its properties. The only exception is `transform.parent`, which you may set in order to change the parent of a GameObject in the scene hierarchy.
4. Transformations should be done with matrix operations only, using `MatrixUtils`. All transforms in this exercise are in world space, relative to the scene origin (0,0,0).

General Guidelines

- Make sure you are using **Unity 6000.2.7f2**
- Make sure that you understand the effect of each part of your code
- Make sure that your code does what it's supposed to do and that your results look the way they should
- Keep your code readable and clean! Avoid code duplication, comment non-trivial code and preserve coding conventions
- Keep your code efficient

Submission

Submit a single `.zip` file containing <u>**only**</u> the following:

- **CharacterAnimator.cs** - Your implementation as described in the exercise.
- **QuaternionUtils.cs** - Your implementation as described in the exercise.
- **`readme.txt`** that includes both partners' IDs and usernames. List the URLs of web pages that you used to complete this exercise, as well as the usernames of all students with whom you discussed this exercise

Deadline

Submit your solution via the course's moodle no later than **Tuesday, November 11 at 23:55**.

Late submission will result in 2^{N+1} points deduction where N is the number of days between the deadline and your submission. The minimum grade is 0, saturday is excluded.

Part 0 / Setup

1. Download the exercise zip file from the course Moodle website and unzip it somewhere on your computer.
2. In Unity Hub, go to *Projects* and click the *Add* button on the top right. Select the folder that you have downloaded.
3. Open the project. Once Unity is open, double click the MainScene to open it.

Part 1 / Building the Skeleton - Joints

- In this part you will implement the function `CreateJoint`. This function creates and positions a GameObject representing the given BVHJoint, then recursively creates objects for its child joints.
- First, add this line in the `Start` function. This will create our skeleton, starting from the root joint at the origin (0, 0, 0):

  ```
  CreateJoint(data.rootJoint, Vector3.zero);
  ```

- Now implement the function as follows. It receives 2 parameters:

  `BVHJoint` **`joint`** - the joint that will be drawn, along with any child joints it may have

  `Vector3` **`parentPosition`** - the 3D position of the parent of the given joint

- The function returns the created `GameObject`.

1. Initialize the `BVHJoint`'s gameObject field (`joint.gameObject`) to an empty `GameObject`, and name it according to the joint's name.
2. Use `CreatePrimitive` to create a sphere representing the joint. Make the sphere GameObject a child of `joint.gameObject` in the scene hierarchy by setting its `transform.parent` to the joint's transform component.
3. Construct and apply a transformation matrix that will scale the sphere by a factor of 2. If the joint's name is `"Head"`, scale the sphere by a factor of 8 instead.
4. Using `MatrixUtils`, construct a translation matrix $\mathbf{T}$ that positions `joint.gameObject` at the correct location in 3D space. As described in the TA slides, the location is determined according to the joint's parent position and the given joint's offset.
5. Apply the matrix to `joint.gameObject` using `MatrixUtils.ApplyTransform`.
6. Apply this recursively to all child joints. You should now be able to see the spheres creating the general shape of a human in 3D space in play mode.

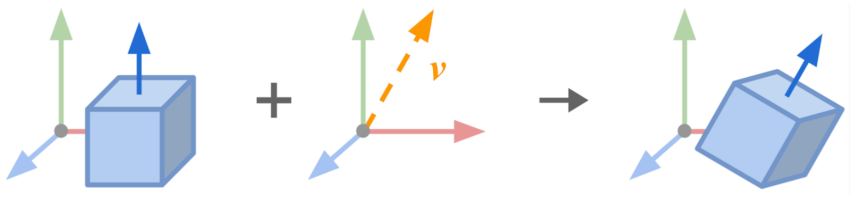
You have finished creating the joints of the skeleton. In parts 2-3 we will implement some helper functions that will allow us to connect these joints with bones.

Part 2 / Rotations

- In this part you will implement the function `RotateTowardsVector`. The function receives 1 parameter:

  `Vector3` **v** - a vector representing the new up direction, after the rotation

- The function returns a `Matrix4×4` representing a rotation that aligns the up vector (aka the y direction (0, 1, 0)) of an object with a given arbitrary vector. This means that the "top" of a transformed object will face the direction of the given $\boldsymbol{v}$:



  In other words, we want to find a rotation matrix $\mathbf{R}$ such that $\mathbf{R}(0, 1, 0)^{\mathrm{T}} = \boldsymbol{v}$.

1. Normalize the given direction vector $\boldsymbol{v}$.

2. Construct and return the rotation matrix $\mathbf{R}$. Use Unity's `Mathf` library as well as `MatrixUtils`. Remember that `MatrixUtils` uses degrees rather than radians.

   Hint: review the TA slides and think how to construct $\mathbf{R}$ from different rotations about the coordinate axes. To test your code, make sure the matrix you get satisfies the equation $\mathbf{R}(0, 1, 0)^{\mathrm{T}} = \boldsymbol{v}$, by using `Matrix4×4`'s `MultiplyVector`.

- In `MatrixUtils` you are given a different implementation of the function `RotateTowardsVector`, using quaternions. You may use that function instead of this one to test other parts of your code. However, note that the solution to $\mathbf{R}(0, 1, 0)^{\mathrm{T}} = \boldsymbol{v}$ is not unique, so the matrix returned from `MatrixUtils.RotateTowardsVector` might be different to the one you get.

Part 3a / Transforming a Cylinder

1. In this part you will implement the function `CreateCylinderBetweenPoints`. The function creates a cylinder GameObject "connecting" two given points in 3D space and returns it.

2. The function receives 3 parameters:

   `Vector3` **`p1`** - first point from which to draw the cylinder

   `Vector3` **`p2`** - second point at which the cylinder should end

   `float` **`diameter`** - width (diameter) of the cylinder to be drawn

3. Use [`CreatePrimitive`]() to create the cylinder. The function creates a cylinder with a height of 2 units and diameter 1 unit, centered at the origin (0, 0, 0).

4. Use `MatrixUtils` to construct a transformation matrix $\mathbf{M} = \mathbf{TRS}$ for the cylinder:

   - $\mathbf{T}$ - Translation matrix. Use it to move the cylinder to the correct position.

   - $\mathbf{R}$ - Rotation matrix. Use the function `RotateTowardsVector` that you have implemented to orient the cylinder so that its top points from p1 to p2.

   - $\mathbf{S}$ - Scaling matrix. Use it to scale the cylinder to the correct proportions.

5. Apply the matrix to the cylinder using `MatrixUtils.ApplyTransform`.

   Hint: test your code as you write it. You can use [`Debug.DrawLine`]() to see where the cylinder should be drawn. Make sure to set a duration to see the actual line!

Part 3b / Building the Skeleton - Bones

- Finally, we can use the functions we have created to connect all the joints with bones.

1. In the function `CreateJoint`, use `CreateCylinderBetweenPoints` with a diameter of 0.6 to connect all the joints with bones.

2. Make the cylinder GameObject a child of `joint.gameObject` in the scene hierarchy by setting its [`transform.parent`]() to the joint's transform component.

You should now see a full skeleton figure in play mode! Check your code with different BVH files to make sure everything works properly.

Part 4 / Animating the Skeleton

1. Implement the function GetFrameNumber. For a given time, it returns the current frame number, according to the `frameLength` given in the `BVHData`. After `numFrames` have elapsed, the frame number should reset to 0 so that the animation loops.

2. Inside the `Update` function, use the current frame number to update the `CurrFrameData` property from the `BVHData`, then call `TransformJoint` on the root `BVHJoint`. For now, set the `parentTransform` parameter such that no transformation will be applied to the root joint.

3. Next you will implement the function `TransformJoint`. This function transforms the given `BVHJoint` according to the current keyframe's channel data, then recursively transforms its child joints.

4. Receives 3 parameters:

   `BVHJoint` **`joint`** - the joint to be transformed, along with any child joints it may have

   `Vector3` **`parentTransform`** - the parent joint's transformation matrix

   `float[]` **`keyframe`** - the channel data to use in the transforms. An array of float values, organized by channels corresponding to a particular transformation of a particular joint, as specified in the BVH Hierarchy

5. Implement the function `TransformJoint`. For each joint in the skeleton, construct and apply a global transform matrix **M'** as described in the TA slides.

6. You should now see the character moving in playmode! (Make sure to turn on the `animate` flag of the `CharacterAnimator` script component.)

7. Finally, recall that the root `BVHJoint` has positional channel data in every keyframe as well as the regular rotational data. Use this data to move the character around according to the animation.

The skeleton should now be animated nicely. Check your code with different BVH files to make sure everything works properly. However, notice when `AnimationSpeed` is slow, the motion appears choppy and discontinuous. You will fix this in the next part.

Part 5a / Quaternions

1. In this part you will implement various functions to create and manipulate quaternion rotations. Open the file `QuaternionUtils.cs` in the code editor.

   You will be using Unity's `Vector4` class to represent quaternions, where the first 3 coordinates x,y,z are the imaginary dimensions and the fourth coordinate w represents the real part:

   $$(x, y, z, w) \Leftrightarrow w + xi + yj + zk$$

2. Implement the functions `Conjugate`, `HamiltonProduct` and `AxisAngle` as seen in class. Note that you are given an implementation of quaternion multiplication in the method `Multiply`.

3. Implement the function `FromEuler`, which converts a given euler rotation to a quaternion. The function receives 2 parameters:

   `Vector3` **`euler`** - Euler angles around the x, y, z axes

   `Vector3Int` **`rotationOrder`** - The order in which to apply the Euler angles

4. Implement the function Slerp. The function receives 2 parameters:

   `Vector4` **`q1`** - The quaternion from which to start interpolation

   `Vector4` **`q2`** - The quaternion at which to end interpolation

   `float` **`t`** - The proportion of interpolation between `q1` and `q2`, in the range [0,1]

   The function returns a spherical linear interpolation between q1 and q2 given by:

   $$\mathrm{slerp}(q_1,q_2,t) = \frac{\sin((1-t)\theta)}{\sin\theta}\, q_1 + \frac{\sin(t\theta)}{\sin\theta}\, q_2$$

   Where θ is *half* the angle between q1 and q2. In the cases where $\sin\theta = 0$, you may return q1. (what does this mean geometrically?)

5. Make sure to test your code with various inputs.

Part 5a / Interpolation

1. Now you will implement interpolation between animation keyframes. This will allow smooth playback at different speeds.

2. Implement the function `GetFrameIntervalTime`, which returns the proportion of time elapsed between the last frame and the next, in the range [0,1]. Use this function to update the class property `float t` when animating in the `Update` method.

3. Update the `nextFrameData` array from the `BVHData` at each frame in the animation. There is no need to update on the last frame of the animation.

4. In the function `TransformJoint`, use the property `t` along with the channel data of the current and next frame to interpolate the transformations between frames.

    - Make sure to interpolate only when `bool interpolate == true`.

    - You may use the function `MatrixUtils.RotateFromQuaternion` to convert a rotation quaternion to a transformation matrix.

    - You may replace any previous Matrix rotations you have used with a quaternion-based implementation.

5. Finally, you may use Unity's implementation of vector linear interpolation via `Vector3.Lerp` to smoothly interpolate the root joint's position between frames.

6. You should now be able to see the character moving smoothly at any animation speed!

Good luck!

`MatrixUtils` documentation

`void` **`ApplyTransform`**`(GameObject gameObject, Matrix4×4 m)`

Applies the transformation matrix `m` to given `gameObject’s` transform component. Note that this overwrites any previous transformations made to `gameObject`.

`Matrix4×4` **`Translate`**`(Vector3 position)`

Returns a `Matrix4×4` representing a translation to the given `position`

`Matrix4×4` **`Scale`**`(Vector3 scale)`

Returns a `Matrix4×4` representing a scale on each axis according to the given `scale`

`Matrix4×4` **`RotateX`**`(float angleDeg)`

Returns a `Matrix4×4` representing a rotation of `angleDeg` degrees around the X axis

`Matrix4×4` **`RotateY`**`(float angleDeg)`

Returns a `Matrix4×4` representing a rotation of `angleDeg` degrees around the Y axis

`Matrix4×4` **`RotateZ`**`(float angleDeg)`

Returns a `Matrix4×4` representing a rotation of `angleDeg` degrees around the Z axis

`Matrix4×4` **`RotateTowardsVector`**`(Vector3 v)`

Returns a `Matrix4×4` representing a rotation aligning the up vector (0, 1, 0) of an object with the given direction vector `v`. See part 2 for more information.

`Matrix4×4` **`RotateFromQuaternion`**`(Vector4 q)`

Returns a `Matrix4×4` representing a rotation equivalent to the given quaternion `q`. See part 5 for more information.

`BVHData` documentation

`BVHJoint` **`rootJoint`**

Root `BVHJoint` object

`int` **`numFrames`**

Number of frames in the BVH animation

`float` **`frameLength`**

Length of each frame in seconds

`List<float[]>` **`keyframes`**

Keyframe data for animating. Each keyframe contains an array of float values, organized by channels. Each channel corresponds to a particular transformation of a particular joint, as specified in the BVH Hierarchy section

`BVHJoint` documentation

`string` **`name`**

Name of the joint

`Vector3` **`offset`**

Offset of the joint relative to the position of its parent

`Vector3Int` **`rotationChannels`**

Indices of the XYZ rotation channels in each keyframe. For example, the X rotation in the first keyframe is given by: `keyframes[0][joint.rotationChannels.x]`

`Vector3Int` **`rotationOrder`**

Order of XYZ rotations. Represents the multiplication order from left to right - for example, a value of `(1,2,0)` corresponds to the rotation $\mathbf{R}_z\mathbf{R}_x\mathbf{R}_y$ i.e. if `rotationOrder.z == 0`, it means that $\mathbf{R}_z$ should be the first rotation from the left

`Vector3Int` **`positionChannels`**

Indices of the XYZ position channels in each keyframe, in a similar manner to `rotationChannels`. Note that only the root joint has `positionChannels`!

`List<BVHJoint>` **`children`**

List of children `BVHJoint`s

`GameObject` **`gameObject`**

Will hold the GameObject of this joint that you will create

`bool` **`isEndSite`**

Indicates if this joint is an EndSite or not. Note that EndSites do not have `rotationChannels` or `positionChannels`